Scaling the Merge Machinery

Elijah Newren

Palantir Technologies

Affected Commands

The merge machinery (merge-recursive) powers several aspects of git:

- merge
- cherry-pick
- revert
- rebase
- am -3
- stash
- checkout -m

The journey

A few years ago...

Issues starting the journey

- cherry-pick would fail to detect renames and fail to notify about needed merge.renameLimit
- cherry-pick would ignore merge.renameLimit > 32767
- if directory renames involved, files would be left in wrong directory
- people wrote custom purpose scripts to cherry-pick things
- after fixing merge.renameLimit, cherry-picking small patches would take more than 9 minutes.

Quotes

The two most prolific authors of git opining on merge-recursive:

- “[It is] some pretty hairy code. Every time I start to look at it I get confused and can’t remember what breakthrough I thought I was close to making before.” (Jeff King)
- “I’ve written off that code as mostly unsalvageable long time ago.” (Junio Hamano)

Goals

Goals for my rewrite of the machinery are to improve each of:

- Maintainability & understandability
- API Quality (enable new features?)
- Correctness
- **Performance**

Types of performance strategies

I have always enjoyed performance talks; they make me feel smarter:

- ~~Squeezing performance out of the hardware~~
- ~~Applying ideas from other problem domains to new areas~~
- ~~Using clever approximation algorithms to get near solutions~~
- ~~Inventing new algorithms~~

Types of performance strategies

Actual performance strategies used:

- Don’t do unnecessary work
- Don’t redo work
- Don’t redo unnecessary work
- Fudge "unnecessary"

Warning

- Glossing over lots of details
- Simplifications not fully accurate

Three-way content merge

File from branch Side1:

```
...
speak_like_a_pirate(arrrgs);
explore_sea(aye, matey);
shiver(me.timbers);
...
```

Same file from branch Side2:

```
...
speak_like_a_pirate(arrrgs);
explore_sea(me.love[0]);
shiver(me.timbers);
...
```

Correct merge depends on the version in the merge base:

```
speak_like_a_pirate(arrrgs);
explore_sea(plus, plus);
shiver(me.timbers);
```

Which results in the following merge:

```
speak_like_a_pirate(arrrgs);
<<<<<<< HEAD
explore_sea(aye, matey);
=======
explore_sea(me.love[0]);
>>>>>>> branchB
shiver(me.timbers);
```


Three-way content merge

File from branch Side1:

```
...
speak_like_a_pirate(arrrgs);
explore_sea(aye, matey);
shiver(me.timbers);
...
```

Same file from branch Side2:

```
...
speak_like_a_pirate(arrrgs);
explore_sea(me.love[0]);
shiver(me.timbers);
...
```

Correct merge depends on the version in the merge base:

```
speak_like_a_pirate(arrrgs);
explore_sea(plus, plus);
shiver(me.timbers);
```

Shorthand:

```
        path
Base :  hash_orig
Side1:  hash_A
Side2:  hash_B
```

Example:

```
        buccaneer.c
Base :  ba771ed
Side1:  57abbed
Side2:  b1a57ed
```

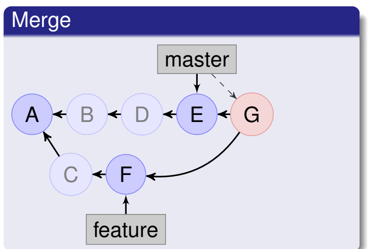
Note: If any two of the hashes match, we can resolve without looking at the contents of the file.

Three-way Merging

```
$ git checkout master
$ git merge feature
```

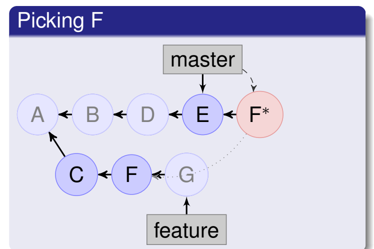
Get three relevant trees, then for each path:

- Get version of path in each tree
- Do three-way content merge



Three-way Merging

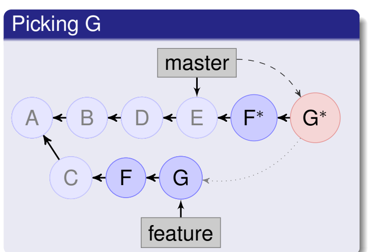
```
$ git checkout master
$ git cherry-pick C..feature
```



Rebasing and reverting are handled similarly to cherry-picking.

Three-way Merging

```
$ git checkout master
$ git cherry-pick C..feature
```



Rebasing and reverting are handled similarly to cherry-picking.

Why renames are important

If a rename is not detected:

```
        buccaneer.c   viking.c
Base:   ba771e5       0000000
Side1:  e5ca185       0000000
Side2:  0000000       defea75
```

Then:

- buccaneer.c: modify/delete conflict
- viking.c: totally new file
- no textual merging

As reported by git status:

```
Changes to be committed:
  new file:   viking.c
Unmerged paths:
  deleted by them:   buccaneer.c
```

If we detect renames on each side of history:

```
         buccaneer.c ⇒ viking.c
Base:    ba771e5
Side1:   e5ca185
Side2:   defea75
Merged:  acc0575
```

Then:

- buccaneer.c: removed
- viking.c: contains merged content

As reported by git status:

EITHER

```
Changes to be committed:
  renamed:   buccaneer.c -> viking.c
```

OR

```
Changes to be committed:
  deleted:   buccaneer.c
Unmerged paths:
  both modified:   viking.c
```


How rename detection works

How does git detect renames? For each side...

Files in Base	Files in given side
README.md	README.md
archery.js	corrupt.js
baseball.js	divine.js
build.log	dull.js
football.js	grand.js
golf.js	lame.js
running.js	

For each pair of files, what percentage of lines are found in both?

	corrupt.js	divine.js	dull.js	grand.js	lame.js
archery.js					
baseball.js					
build.log					
football.js					
golf.js					
running.js					

Matrix of similarity percentages

How rename detection works

Crux of the problem

Rename detection is O($M * N$), where M and N are **huge**.

{M, N} ~ O(combined line count of potential rename {sources, targets})

Optimization 1: Don't redo work

Don't look for a better than perfect match.

Exact renames

Detecting renames

```
void detect_renames_and_copies(...)
{
    ...
    exact_count = find_different_name_same_hash();
    /* Keep all the source files as options for copies! */
    for (dest_path in potential_rename_targets) {
        if (already_paired(dest_path)) continue;
        for (source_path in potential_rename_sources) {
            if (!DETECT_COPIES &&
                already_paired(source_path))
                continue;
            compute_similarity();
        }
    }
    ...
}
```


Optimization 2: Don't do unnecessary work

If you can get the same answer without an expensive computation, skip the expensive computation.

Partial capitulation

If a rename is not detected for the merge:

```
        buccaneer.c  viking.c
Base:   5eac0a57     00000000
Side1:  5eac0a57     00000000
Side2:  00000000     c01055a1
```

Then:

- buccaneer.c: modify/delete conflict
- viking.c: totally new file
- no textual merging

As reported by git status:

```
Changes to be committed:
  new file:   viking.c
Unmerged paths:
  deleted by them:  buccaneer.c
```

Same results whether or not rename is detected by merge machinery.

If we detect renames on each side of history:

```
        buccaneer.c ⇒ viking.c
Base:   5eac0a57
Side1:  5caff01d
Side2:  c01055a1
Merged: ????????
```

Then:

- buccaneer.c: removed
- viking.c: contains merged content

As reported by git status:

EITHER

```
Changes to be committed:
  renamed:   buccaneer.c -> viking.c
```

OR

```
Changes to be committed:
  deleted:   buccaneer.c
Unmerged paths:
  both modified:   viking.c
```


Partial capitulation

If a rename is not detected for the merge:

```
        buccaneer.c  viking.c
Base:   5eac0a57     00000000
Side1:  5eac0a57     00000000
Side2:  00000000     c01055a1
```

Then:

- buccaneer.c: deleted as expected
- viking.c: totally new file
- no textual merging needed

As reported by git status:

```
Changes to be committed:
  renamed:   buccaneer.c -> viking.c
```

Same results whether or not rename is detected by merge machinery.

If we detect renames on each side of history:

```
        buccaneer.c ⇒ viking.c
Base:   5eac0a57
Side1:  5eac0a57
Side2:  c01055a1
Merged: c01055a1
```

Then:

- buccaneer.c: removed
- viking.c: no content merge was needed

As reported by git status:

```
Changes to be committed:
  renamed:   buccaneer.c -> viking.c
```


Partial capitulation – Caveats?

New Strategy

Exclude potential source from rename detection **if** it is unmodified by *other* side of history and parent directory of source file exists on *same* side of history.

Possible problems:

- ~~causes issues for directory rename detection~~
- rename/add conflict looks like add/add
- rename/rename(2to1) conflict looks like rename/add or add/add

"Mis-detected" conflict types:

- ~~Different conflict-related files in the working copy~~
- ~~Different conflict-related entries in the index~~
- Different stdout; reports e.g. `CONFLICT(add/add)` instead of `CONFLICT(rename/add)`

After unifying file collision conflict handling...stdout is only difference.

Partial capitulation – micro or mega optimization?

New Strategy

Exclude potential source from rename detection **if** it is unmodified by *other* side of history and parent directory of source file exists on *same* side of history.

How much does this new strategy help?

A Common Case

$O(M * N) \to O(\emptyset * N)$

Optimization 3: Fudge "unnecessary"

Only do part of the work and accept slightly different results if there are huge cost savings.

Dimensionality Reduction

Fun fact

Over 75% of renames in the linux kernel repository do not change the basename of the file, just the directory in which it is found.

Dimensionality Reduction

Detecting renames...

Files in Base	Files in given side
document.html	build.log
src/blue.css	document.html
src/brown.css	source/blue.css
src/green.css	source/brown.css
src/red.css	source/green.css
	source/orange.css
	source/purple.css
	source/red.css

For each pair of files, what percentage of lines are found in both?

	src/blue.css	src/brown.css	src/green.css	src/red.css
build.log				
source/blue.css				
source/brown.css				
source/green.css				
source/orange.css				
source/purple.css				
source/red.css				

Matrix of similarity percentages

Sparse similarity percentages

Dimensionality Reduction

Improvement

$O(M * N) \rightarrow$ O((M-B)*(N-B) + B)

If enough matching basenames...

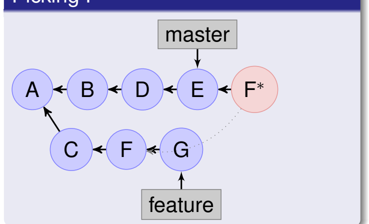
$O(M * N) \rightarrow$ O(minimum(M, N))

Optimization 4: Don't redo unnecessary work

When repeatedly merging, re-use previous rename detection results.

Remembering previous work

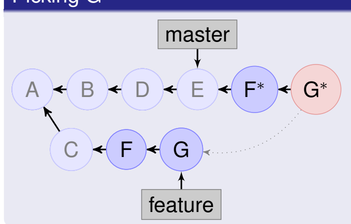
Picking F



$$\left\{\begin{array}{lll} & \text{buccaneer.c} & \text{viking.c}\\ C: & \texttt{c0a575} & \texttt{000000}\\ E: & \texttt{000000} & \texttt{b1ade5}\\ F: & \texttt{befa11} & \texttt{000000}\end{array}\right\} \rightarrow \left\{\begin{array}{ll} & \text{buccaneer.c} \Rightarrow \text{viking.c}\\ C: & \texttt{c0a575}\\ E: & \texttt{b1ade5}\\ F: & \texttt{befa11}\\ F^*: & \texttt{1007ed}\end{array}\right\}$$

Remembering previous work

Picking G



$$\begin{array}{ll} & \text{buccaneer.c} \Rightarrow \text{viking.c}\\ C: & \texttt{c0a575}\\ E: & \texttt{b1ade5}\\ F: & \texttt{befa11}\\ \hline F^*: & \texttt{1007ed}\end{array}$$

$$\left\{\begin{array}{lll} & \text{buccaneer.c} & \text{viking.c}\\ F: & \texttt{befa11} & \texttt{000000}\\ F^*: & \texttt{000000} & \texttt{1007ed}\\ G: & \texttt{a70115} & \texttt{000000}\end{array}\right\} \rightarrow \left\{\begin{array}{ll} & \text{buccaneer.c} \Rightarrow \text{viking.c}\\ F: & \texttt{befa11}\\ F^*: & \texttt{1007ed}\\ G: & \texttt{a70115}\\ G^*: & \texttt{fab1ed}\end{array}\right\}$$

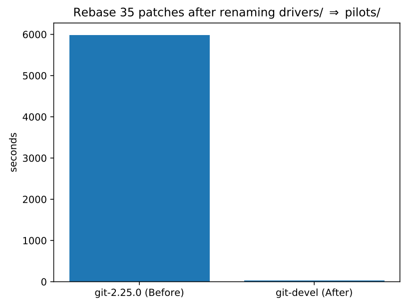
But wait, there's more!

- Avoid accidentally quadratic behavior
- Restructure to eliminate quasi-quadratic index insertion and removal
- Fewer tree traversals
- Extend "partial capitulation" ideas from *file* renames to *directory* renames
- Avoid updating the index or working tree if not needed
  - Helps with new sparse-checkoutsparse-checkout command
  - Accelerates rebases and cherry-picks
  - Avoids unnecessary recompilationAvoids unnecessary recompilation after a rebase
- ...and a few other minor improvements

Testcase

- Linux kernel
- Rebase or cherry-pick hwmon-updates (35 patches) from 5.5 $\rightarrow$ 5.4
- Very few renames involved; only takes 50% of execution time
- Speedup factor of 3 (optimized more things than renames)
- What if we checkout 5.4, and rename `drivers/` $\Rightarrow$ `pilots/`, and then rebase or cherry-pick those 35 patches on top ?

Results



Results

Reproduce these numbers:

https://github.com/newren/git/blob/git-merge-2020-demo/README.md

The journey, redux

Issues starting the journey

- cherry-pick would fail to detect renames and fail to notify about needed merge.renameLimit
- cherry-pick would ignore merge.renameLimit > 32767
- if directory renames involved, files would be left in wrong directory
- people wrote custom purpose scripts to cherry-pick things
- after fixing merge.renameLimit, cherry-picking small patches would take more than 9 minutes.

When I told folks a few years ago that "You don't need these special scripts to cherry pick things; just set merge.renameLimit to something higher," they responded that merge.renameLimit didn't work.

I didn't believe them.

My efforts in this area, including this performance work, represent my attempt to continue to not believe them. :-)